ASSIGNMENT 5

Handout: **Tuesday, 22 November 2016**
Due: **11:30 am, Thursday, 1 December 2016**

GOALS:

- To understand better concurrent programming in Java;
- To practice using monitor operations to coordinate the access to shared objects;
- To learn about `ReentrantLock` and `Condition` objects;
- To get more familiar with the IntelliJ Idea IDE;

1. BANKACCOUNT USING MONITOR OPERATIONS

Write a class `BankAccount` so that the following code snippet compiles and always prints out 20. Coordinate the `withdraw` and `deposite` operations on a shared bank account using `synchronized` methods or blocks and `wait`/`notify`/`notifyAll`. Pay attention to method calls that may trigger an `InterruptedException`.

You may assume the arguments of method `withdraw` and `deposite` are always non-negative.

```
public class Withdrawer implements Runnable{
  private BankAccount ba;
  private int amount;

  public Withdrawer(BankAccount ba, int amount){
    this.ba = ba;
    this.amount = amount;
  }

  public void run() {
    for (int i = 0; i < 10; i++){
      ba.withdraw(amount);          // Note that a withdraw is only allowed when the
                                    // balance is greater than the amount to withdraw
    }
  }
}

public class Depositor implements Runnable{
  private BankAccount ba;
  private int amount;

  public Depositor(BankAccount ba, int amount){
    this.ba = ba;
    this.amount = amount;
  }
```

```
  public void run() {
    for(int i = 0; i < 10; i++){
      ba.deposit(amount);                    // A deposit is always allowed.
    }
  }
}

public class WithdrawerDepositor {
  public static void main(String[] args) {
    BankAccount ba = new BankAccount(0); // initialize the bank account to have balance 0
    Thread withdrawer = new Thread(new Withdrawer(ba, 3));
    Thread depositor = new Thread(new Depositor(ba, 5));
    withdrawer.start();
    depositor.start();
    try {
      withdrawer.join();
      depositor.join();
    }
    catch(InterruptedException e){}

    System.out.println(ba.getBalance());
  }
}
```

2. BankAccount Using ReentrantLock and Condition

In this task, you need to use `ReentrantLock` and `Condition` objects to re-implement the `BankAccount` class from Task 1 and achieve the same result.

Read the Java docs about `ReentrantLock` and `Condition` at the following addresses:
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantLock.html
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Condition.html

What to do:

Implement class `BankAccount` using two different sets of APIs for concurrency (10 points each).

What to hand in:

The two `BankAccount` classes.